Connor Lack

COS30018 - Option B - Task 4: Machine Learning 1

As the model has progressed into version v0.3, task 4 required the implementation of a better method for constructing the Deep Learning model. This meant altering the prior code which constructed the deep learning model network manually, and creating a function that has the minimum requirement as input: The number of layers, the size of each layer, and the layer name. It was also asked of us to explore different DL networks and experiment with hyperparameter configurations.

In order to achieve this task, a new function, *'build model',* was incorporated into the code which focused on building a sequential model blueprint that would be used for model experimentation for identifying which RNN and hyperparameters have the best accuracy.

Arguments for '*build model'* included:

- sequence_length: time steps in each input sequence
- n_features: features in the input data
- cell (keras.layers.RNN): RNN cell used (LSTM, GRU, etc.)
- units: Neurons in each recurrent layer
- n_layers: Number of layers
- dropout: Dropout rate for layers
- optimizer: Optimization algorithm
- loss: Loss function
- bidirectional: Bidirectional Y/N

'*build model'* returned:

model (keras.Sequential): Compiled model

```python
def build_model(sequence_length, n_features, cell=LSTM, units=256, n_layers=2, dropout=0.3,
                optimizer='rmsprop', loss='mean_absolute_error', bidirectional=False):
    """
    Builds a sequential model for stock price predictions based on parameters.
    """

    # Defines the shape of the input data, which will depend on the length and number of features
    input_shape = (sequence_length, n_features)

    # Initializes the Sequential model (a linear stack of layers)
    model = Sequential()

    # For loop that adds specified amount of layers (n_layers) to the model
    for i in range(n_layers):
        # If the current layer is the last layer
        is_last_layer = (i == n_layers - 1)
        # 'return_sequences' is equal to True except for the last layer
        # This allows each of the time outputs to be fed to the next layer
        return_seq = not is_last_layer

        # Specify the input shape for the first layer
        if i == 0:
            layer_input_shape = input_shape
        else:
            layer_input_shape = None    # Keras can sort it out

        if bidirectional:
            # Adds a Bidirectional layer if needed
            # Allows cell to process input sequences in both directions
            model.add(Bidirectional(cell(units, return_sequences=return_seq), input_shape=layer_input_shape))
        else:
            # Adds the specified cell (LSTM, GRU, SimpleRNN, etc.) to model
            model.add(cell(units, return_sequences=return_seq, input_shape=layer_input_shape))

        # A dropout layer after each recurrent layer to reduce overfitting
        # Randomly setting some input units to 0 at each update during training
        model.add(Dropout(dropout))

    # Dense layer that produces the final output
    # Linear activation is used since a continous value is being predicted
    model.add(Dense(units=1, activation='linear'))

    # Compiles the model with the specified optimizer and loss function
    # Able to monitor the model's performance with metrics such as 'mean_absolute_error'
    model.compile(optimizer=optimizer, loss=loss, metrics=['mean_absolute_error'])

    # Returns model, ready for training
    return model
```

Figure 1 - Build Model function

The function begins by defining the input_shape, which is based on the sequence_length (number of time steps in each input) and n_features (number of features per time step). Time step refers to the number of data points the models looks at for a prediction, in this instance, it would be prediction days for the data.

It then enters a for loop for n_layers (specified layer count), where for each layer it checks if it's the last layer in order for return_sequences=False in order to output a single vector.

The input_shape is then determined by the first layer, whilst keras sorts out the following layers to infer the shape. The if statement for the bidirectional adds either a bidirectional layer or not.

A dropout layer is then added after each recurrent layer, preventing overfitting through random disabling of neurons during training.

After recurrent layers, a dense layer is added to produce the final output, which is the stock's predicted price. The model then gets compiled with the optimizer and loss function outlined, readying it for training. 'mean_absolute_error' allows for the user to see monitoring performance.

Additionally, new code was added to the main function in the program in order to enable experimentation with the different deep learning models and configurations of the hyperparameters. While in previous versions of the stock prediction program, only a single LSTM model could be configured, now an experiments list is defined, which contains dictionaries that detail the unique parameters for each model.

This includes the type of RNN cell (LSTM, GRU, etc.), unit numbers, number of layers, dropout rate, bidirectional, epochs, batch size, optimizer, and loss function.

The for loop iterates over the list, and calls the build_model function for each experiment to construct the model with the specific parameters input. model.fit trains the model after it's built.

The prepare_data function is then utilised to prepare the test data. Predictions are made, then both the predicted and proper prices are inverse transformed using scalers back to their original scale.

```python
# Update the input shape for model
sequence_length = x_train.shape[1]  # Time steps in each sequence
n_features = x_train.shape[2]       # Features in each time step

# Configurations for each experiment to compare the model performances
experiments = [
    {
        'cell': LSTM,
        'units': 50,
        'n_layers': 3,
        'dropout': 0.2,
        'bidirectional': False,
        'epochs': 25,
        'batch_size': 32,
        'optimizer': 'adam',
        'loss': 'mean_squared_error'
    },
    {
        'cell': GRU,
        'units': 64,
        'n_layers': 2,
        'dropout': 0.3,
        'bidirectional': True,
        'epochs': 30,
        'batch_size': 16,
        'optimizer': 'adam',
        'loss': 'mean_absolute_error'
    },
    {
        'cell': SimpleRNN,
        'units': 128,
        'n_layers': 4,
        'dropout': 0.1,
        'bidirectional': False,
        'epochs': 20,
        'batch_size': 64,
        'optimizer': 'adam',
        'loss': 'mean_squared_error'
    }
]

# Loops over experiments to train / test the model configurations
for idx, exp in enumerate(experiments):
    print(f"\n--- Experiment {idx+1}: {exp['cell'].__name__} Model ---")

    # Builds the model with the specified parameters in the experiment above
    model = build_model(
        sequence_length=sequence_length,      # sequence length
        n_features=n_features,                # no. features per time step
        cell=exp['cell'],                     # RNN cell
        units=exp['units'],                   # no. neurons per layer
        n_layers=exp['n_layers'],             # no. recurrent layers
        dropout=exp['dropout'],               # dropout rate
        optimizer=exp['optimizer'],           # optimizer
        loss=exp['loss'],                     # loss functions
        bidirectional=exp['bidirectional']    # bidirectional layers Y/N?
    )

    # Trains the model with training data x_train, y_train
    model.fit(
        x_train, y_train,                     # training features, training labels
        epochs=exp['epochs'],                 # no. epochs
        batch_size=exp['batch_size'],         # no. samples per gradient
        validation_split=0.1,                 # training data used for validation
        verbose=1                             # verbosity mode (progress bar)
    )

    # Prepares test data using same function as training data
    x_test, y_test = prepare_data(test_data, feature_columns, PREDICTION_DAYS)

    # Trained model predicts stock prices on test data
    predicted_prices = model.predict(x_test)

    # Inverse transform predicted prices using the 'Close' scaler
    predicted_prices = scalers['Close'].inverse_transform(predicted_prices)

    # Reshape and inverse transform the actual prices to og scale
    actual_prices = y_test.reshape(-1, 1)
    actual_prices = scalers['Close'].inverse_transform(actual_prices)

    # Flattens the arrays to one dimension for plotting
    predicted_prices = predicted_prices.flatten()
    actual_prices = actual_prices.flatten()
```

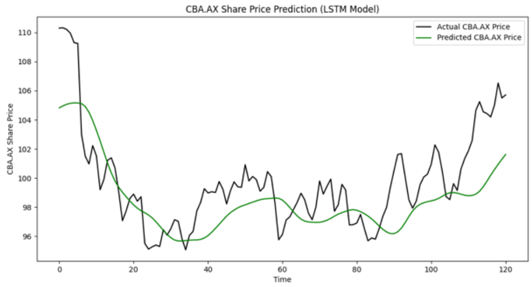
Figure 2 - Adjusted main function

Connor Lack

Testing with the experiments

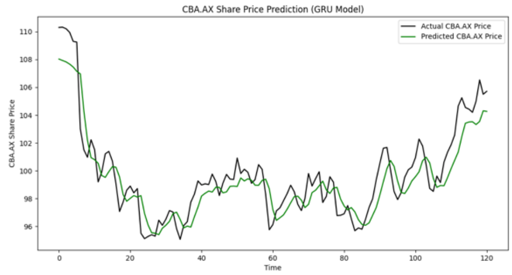
The following hyperparameters were utilised:

```
experiments = [
        {
            'cell': LSTM,
            'units': 50,
            'n_layers': 3,
            'dropout': 0.2,
            'bidirectional': False,
            'epochs': 25,
            'batch_size': 32,
            'optimizer': 'adam',
            'loss': 'mean_squared_error'
        },
```



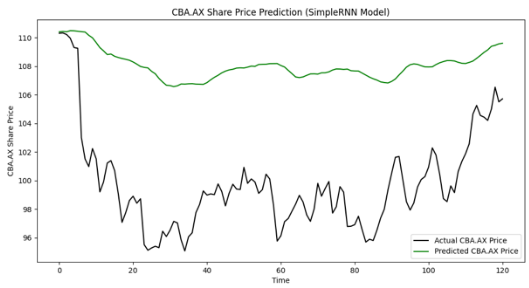
The LSTM model fairly accurately predicted stock prices well, as it was able to understand the general trends of the data. This is indicated through the smooth and less volatile predictions. However, this also means it struggled with sharp fluctuations.

```
        {
            'cell': GRU,
            'units': 64,
            'n_layers': 2,
            'dropout': 0.3,
            'bidirectional': True,
            'epochs': 30,
            'batch_size': 16,
            'optimizer': 'adam',
            'loss': 'mean_absolute_error'
        },
```



The GRU model's predictions seem to be the best of the three, containing smooth predictions, whilst also following the volatile spikes in the data fairly accurately, resulting in better predictions in price fluctuations. However, it can be stated that the model has been overfitted, as it tends to stick too close to the training dataset, failing to generalize. This could've resulted from too little training size.

```
        {
            'cell': SimpleRNN,
            'units': 128,
            'n_layers': 4,
            'dropout': 0.2,
            'bidirectional': False,
            'epochs': 40,
            'batch_size': 64,
            'optimizer': 'adam',
            'loss': 'mean_squared_error'
        }
```



The SimpleRNN model resulted in a large mismatch between the predicted and actual data. This might be a result of the SImpleRNN architecture, which is more incapable of handling complex sequential dependencies.

Connor Lack

References / Reading (APA 7)

1. Keras. (n.d.). LSTM layer. https://keras.io/api/layers/recurrent_layers/lstm/
2. Keras. (n.d.). Model training APIs. https://keras.io/api/models/model_training_apis/
3. TensorFlow. (2023, September 27). Time series forecasting. https://www.tensorflow.org/tutorials/structured_data/time_series
4. TensorFlow. (2023, October 2). Recurrent neural networks (RNN) with Keras. https://www.tensorflow.org/guide/keras/working_with_rnns
5. Brownlee, J. (2020, August 20). Time series prediction with LSTM recurrent neural networks in Python with Keras. Machine Learning Mastery. https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/